A client-server protocol for local area networks

The layered approach to the design of networked systems is generally agreed to result in cleaner and more maintainable products. In addition, the specification of standards for "open" systems contributes to the portability and compatibility of various pieces of the architecture in such a way that implementors may "mix and match" layers that are appropriate to their systems.

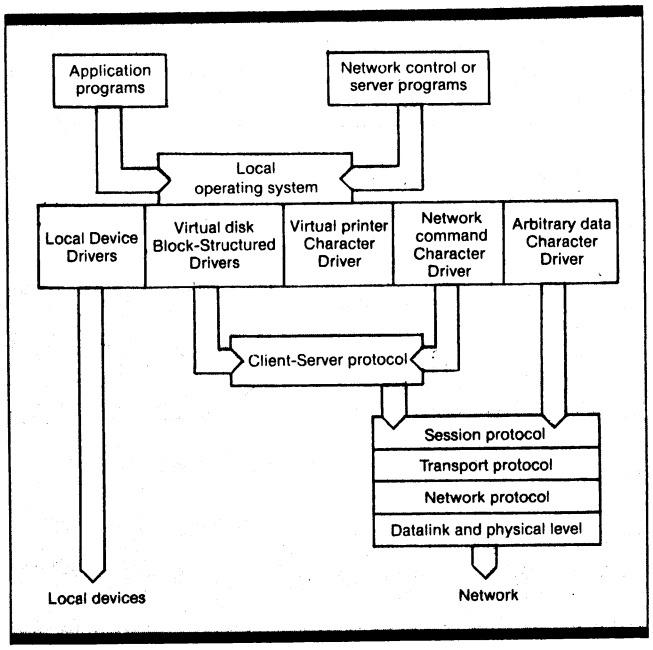
Nestar Systems uses various existing standards for the lower-level network protocol corresponding to ISO-OSI layers one through four. For several years, however, it has successfully used various versions of a client-server protocol designed specifically for efficient implementation on microcomputer systems. Parts of the protocol, and the reasons for it, are described here, along with the invitation for others to use this design or contribute to extensions.

The system environment consists of computers from many manufacturers interconnected via a local area network, which may in turn be connected to other networks. Each computer may be considered either a "server" station which supplies the system with the shared resources, or a "client" station which makes use of those services. The division is a logical one only; the same computer clearly may be a server or client at different times, and may even be both at the same time.

Nestar uses various personal computers (IBM PC, IBM PC XT, Apple II, Apple III, etc.) as both client and server stations, and also manufactures higher performance server stations (Plan 3000, Plan 4000) based on the 68000 microprocessor. When used as a client station, the personal computer may be executing under any of several manufacturer-supplied operating systems, including the p-System, MS-DOS, Apple SOS, Apple DOS, and CP/M.

With such a diverse collection of hardware and software to support, it is obviously critical that the architecture of all layers of the network system be machine independent. Part of this constraint includes making sure that a useful subset can be implemented on small machines with primitive operating systems. For the Apple II under the DOS operating system, for example, an implementation of the protocols described here fits in 2 kbytes of machine code for the 6502 microprocessor.

Leonard J. Shustek, Vice President, Research and Development
Nestar Systems Inc.
2585 East Bayshore Road
Palo Alto, Calif. 94303


Application programs
Network control or server programs
Local operating system
Local Device Drivers
Virtual disk Block-Structured Drivers
Virtual printer Character Driver
Network command Character Driver
Arbitrary data Character Driver
Client-Server protocol
Session protocol
Transport protocol
Network protocol
Datalink and physical level
Local devices
Network

1. The client server protocol supplies virtual device support transparently to the local OS.

Which layers go where. The first four layers of the protocol must provide a reliable delivery service for messages of any size between any two interconnected computers. Nestar combines (see "Nestar Plan Series Network System layering") the Datapoint ARCNET for the physical and datalink layers, with the Xerox Network Systems (XNS) Internet and Sequenced Packet Protocol (SPP) for the network and transport layers.

At higher levels, the protocols must strike a delicate balance that preserves layer independence yet allows for efficient implementation. In addition, it must be able to supply basic network services (device sharing and command processing) in a transparent fashion beneath an existing operating system which may not permit multiple processes.

The session-control protocol provides a way to structure the messages into connections. In the full implementation, multiple simultaneous connections are allowed; in subset implementations only a single connection is supported. To maximize efficiency, the session protocol should avoid requiring special transport-level messages and should depend on the context of the conversation as much as possible.

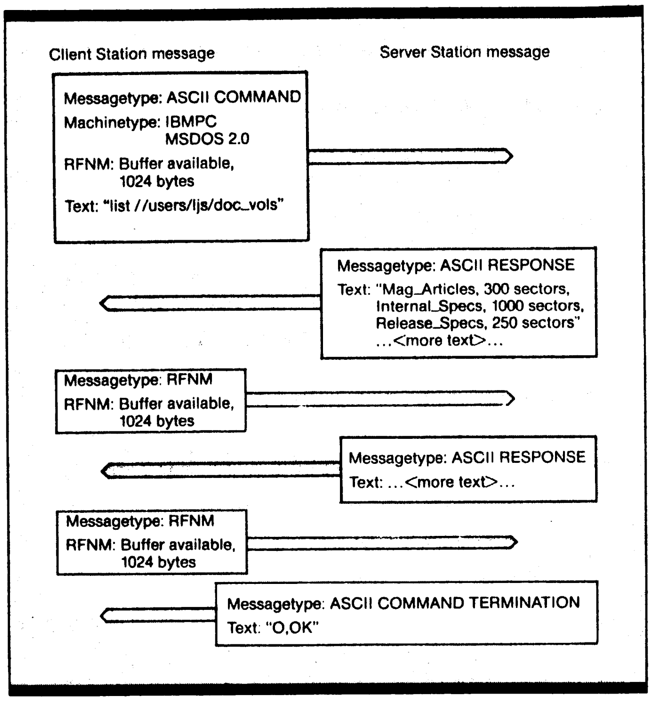
The client-server protocol provides a mechanism for client stations to make requests of servers and exchange the data necessary to complete that request. Included in the specification of this protocol is the format of control blocks for two of the most important service requests for network file servers: arbitrary text command execution, and virtual disk access.

Although the Transport and session levels can provide full-duplex transmission, connections used by the client-server protocol are usually short-lived and consist of an alternating bidirectional message stream. Each process knows from the semantics of the conversation whether it is expected to receive or transmit; since lower levels have guaranteed reliable transmission (or have reported a failed connection), there is not ambiguity. Server stations may have many such connections in progress simultaneously, of course.

The basic and most important use for the client-server protocol is to supply virtual device support in a transparent manner to the local operating system (Fig. 1). For many environments, four types of network drivers are provided:

- Several block-structured devices representing virtual disks managed by the file server
- An output device representing a virtual printer whose data will be processed by the print server
- A single bidirectional device used to send commands and receive responses from servers, or to communicate with the network-layered software running beneath the operating system.
- A bidirectional device used to exchange arbitrary data with other computers. This provides simplified access to the transport layer.

By using the operating system's device driver conventions whenever they are available, application programs then have convenient access to network facilities using the standard I/O facilities of high-level languages. For additional sophisticated applications, involving multiple connections and interrupt-driven code, direct access to the network software at the session and transport level is also provided which bypasses the often restrictive local operating system.


Client Station message
Server Station message
Messagetype: ASCII COMMAND
Machinetype: IBMPC MSDOS 2.0
RFNM: Buffer available, 1024 bytes
Text: "list //users/ljs/doc_vols"
Messagetype: ASCII RESPONSE
Text: "Mag_Articles, 300 sectors, Internal_Specs, 1000 sectors, Release_Specs, 250 sectors" ...<more text>...
Messagetype: RFNM
RFNM: Buffer available, 1024 bytes
Messagetype: ASCII RESPONSE
Text: ...<more text>...
Messagetype: RFNM
RFNM: Buffer available, 1024 bytes
Messagetype: ASCII COMMAND TERMINATION
Text: "O,OK"

2. In the command/Response model, conversations consist of a client command and server response.

Client-server conversations: commands. The simplest client-server conversation consists of a textual command sent by the client and subsequent responses sent by the server. An example might be a request to list part of the file server's hierarchical directory of files.

All messages transmitted using the client-server protocol use the datastream type, which is separate from the body of the data, to indicate how the data is to be interpreted. A textual command begins (Fig. 2) with the transmission of a message of type "ASCII COMMAND", which serves both to initiate a connection at the session level and to transmit a request at the client-server level. It describes, with several encoded numbers, hardware and software of the originating station as well as the text of the command.

In addition, the request contains a small object called a "Request For Next Message", or RFNM. The RFNM is used as part of session-level flow control in several message types to inform the other participant of how much buffer space is available. Although flow control also exists at lower levels, flow control at the session level is vitally important for high-performance implementations. It, combined with the datastream type which is not part of the body of the data, allows data to be directly transferred from the packet buffers at the datalink level to buffers at the operating system or application level.

After the request has been sent, the server responds—perhaps after some delay—with as much text response to the command as it wishes, provided that the buffer size specified in the client's RFNM is not exceeded. The client station responds with a message containing only a RFNM, to indicate both that it is ready for the next part of the response and how much it can accept. The server continues until there is no more response text, at which time it sends back a message summarizing the success or failure of the command. The connection is broken at this point and no further messages are sent.

Note the efficiency of the exchange: There are no special messages for session control, and a single message is sent at a time in each direction. In addition, since the messages alternate, no additional packets at the transport level are required since acknowledgments are piggybacked on the first packet of the message in the other direction; the only exception is a single additional packet at the end to acknowledge the last message.

The isolation of the final response as a message with a separate datastream type is convenient so that the client-station process need know nothing about the expected response sequence. At the application level it can display, buffer, or even ignore the intermediate responses if the principal effect of the command is successful execution by the server.

In addition to textual responses directed to the application level, it is often important for the server to communicate other information to lower levels, preferably in encoded format. For example, a MOUNT command sent to associate a volume in a file server's virtual disk hierarchy with a particular local drive may require modification to various local operating systems or device driver tables. In that case the server, before sending the ASCII COMMAND TERMINATION message, sends a DEVICE INFORMATION MESSAGE (DIM) that describes the drive on which the virtual disk was mounted, the access rights, the size, and other useful information. In addition to supplying information that the client station could get no other way, it also allows the client-station device driver to avoid parsing textual commands that originate at the applications level since the server will need to do so anyway, and can send back encoded information necessary to change local state tables.

Client-server conversations: I/O transfers. The second important example of a client-server exchange is the request made by the Client's local operating system for data transfer to or from a network virtual device such as a printer or disk. It begins, as with Command requests, by the client sending a message, called an Input/Output Block (IOB), to both establish the connection and indicate the nature of the request (Fig. 3). At this point the protocol varies, depending on whether the request is to read a disk (transfer from server to client) or write a disk (transfer from client to server).

Reading disk data is similar in some respects to command execution. The case of writing to the disk is different because the direction of data and RFNMs reverse: the client waits until the server indicates, by sending a RFNM, that data may be transmitted. The client transmits data not exceeding the amount specified in the RFNM, and then waits again for a message. The possible server responses are RFNM (indicating more data can be sent), or IOB TERMINATION (indicating that the exchange is complete and returning operating and device status).

Efficient implementation. Errors can cause the normal protocol to be aborted at any time. Yet because efficient implementation requires that disk data be transmitted directly into the application data buffer, it is not acceptable for an error message to be transmitted instead of data when an abort is required. To resolve this dilemma, a zero-length message is transmitted whose datastream type is ABORT. Since the datastream type is not transmitted to the buffer, the buffer is not contaminated and yet the receiving process knows that the normal protocol has been replaced by the error protocol, and prepares to receive the error

Nestar PlanSeries Network system layering

Layer Name	Layer Number	Purpose	Typical Objects/Operations	Implementation
Application	—	Many: database, graphics, decision support, network control	Application data	(various)
Local O.S.	—	Local process and device control	File access operations	(various)
Client/Server	—	Remote service and data requests	Service requests, applications data	Nestar PLAN protocols
Session	5	Establish and control multiple connections	Connection requests	Nestar PLAN protocols
Transport	4	Message assembly; reliable error recovery	Messages	Xerox XNS
Network	3	Address mapping; gateway interface	Internet packets	Xerox XNS
Datalink	2	Packet transmission	Packet	Datapoint Arcnet
Physical	1	Electrical and mechanical cable interface	Bits	Datapoint Arcnet

message into an appropriate buffer. The datastream type, which is a standard part of the XNS Sequenced Packet Protocol, is an elegant device that efficiently accommodates rare events in such a way that the speed of normal data transfer is not affected.

The sticky problems. As always, the problems in designing protocols arise in the handling of error conditions, not in seeing that everything works in the normal case. There are several classes of errors that must be carefully considered in the design and implementation; the following are a few such errors.

TIMEOUTS: Although the Transport level guarantees reliable transmission if it occurs, it cannot guarantee that transmission will occur. It is possible that the physical link may break, one of the participant computers may malfunction, or delays may be unacceptable. Timeouts are error events representing failed transmissions that occur at lower levels but must be propagated to higher levels for processing.

There are really two problems: (1) How to choose timeout values, and (2) how to respond to timeouts. Since most timeouts are implemented by the transport level but are processed by the session and client-server levels, control of timeout values is in the interface between them. The best scheme is for transport to choose appropriate defaults which can be modified as necessary by other levels.

The response to timeouts depends on the level of service to be provided. At some relatively high level, the user or application program can specify how persistent the attempt to obtain service should be, and that can be used to modify the timeout or retry values. In general, the specification of timeout algorithms that maximizes performance yet minimizes failed operations is not simple, and no one set of values can satisfy all clients.

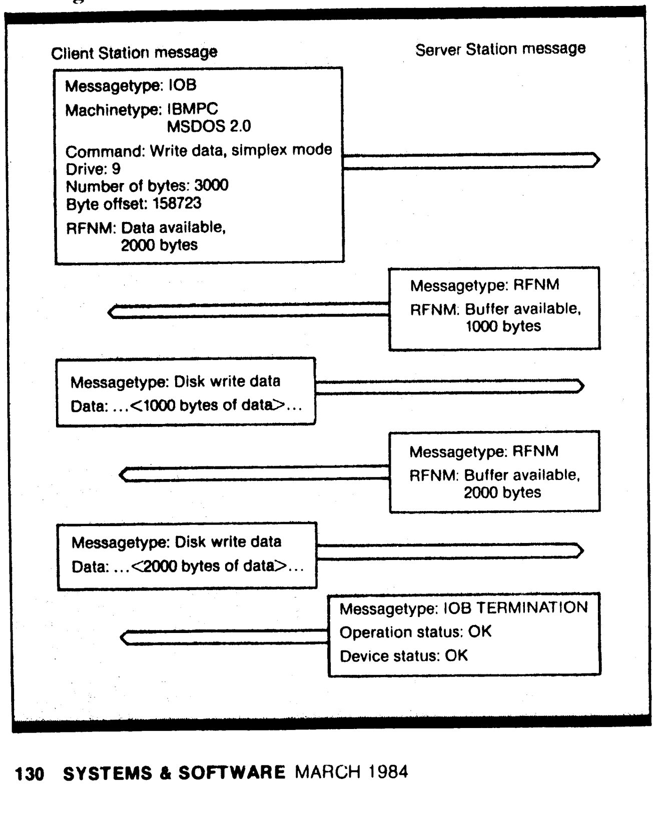
ABORTS: A client-server command may be aborted by either participant when a synchronous error occurs (an attempt to access a write-protected virtual disk, for example), or an asynchronous event occurs (a keyboard abort key, when enabled). It is acceptable to send a zero-length ABORT message in place of the normal message at any point in the protocol; the ABORT message is then followed by a data message with information about the cause of the abort.

NON-RETRYABLE COMMANDS: Because processing occurs by both client and server as messages of the protocol are exchanged, care must be taken in the style of error recovery at the transport level. If, for example, the transport level's recovery from a lost packet or broken connection is to ask the session level to restart the connection, then all partially completed processing must either be undone or permitted to be redone. An example of a command which cannot be redone is a file-creation request; if the connection is broken after the initial request but before the final response message, and if the client station reissues the command in such a way that the server does not recognize it as a duplicate, it will fail. It will fail because the file already exists when the command is processed the second time. This is a complex problem in general and can be handled by carefully choosing the error recovery strategies at all levels, by command sequence numbering, or by adopting a "transaction commit" strategy from database technology.

Maintaining high performance: If not designed and implemented carefully, a highly-layered system is likely to be outstandingly inefficient. To avoid this, it is crucial that data not be copied each time a layer encapsulates or decapsulates information from neighboring levels. It is possible, given the appropriate design for both hardware and software, for messages to be processed by all levels but for the data to move only once from the network packet buffer to the applications level buffer. Having done so, the process-to-process network transmission speeds through the Nestar 2.5-Mbits/s ARCNET and exceeds most implementations using more expensive network hardware that is in theory three to four times faster.

Making it all work. Other parts of the client-server protocol have been defined to accommodate special functions. For example, a "Boot protocol" exists to allow diskless client stations to broadcast a request for assistance in downloading an operating system by any available fileserver. After the local operating system has control, the normal processes for locating and communicating with servers can take

3. In the disk-write model, the client sends an I/O block message to transfer data from client to server.


Client Station message
Server Station message
Messagetype: IOB
Machinetype: IBMPC MSDOS 2.0
Command: Write data, simplex mode
Drive: 9
Number of bytes: 3000
Byte offset: 158723
RFNM: Data available, 2000 bytes
Messagetype: RFNM
RFNM: Buffer available, 1000 bytes
Messagetype: Disk write data
Data: ...<1000 bytes of data>...
Messagetype: RFNM
RFNM: Buffer available, 2000 bytes
Messagetype: Disk write data
Data: ...<2000 bytes of data>...
Messagetype: IOB TERMINATION
Operation status: OK
Device status: OK

place. Another extension of the protocol allows a client station to duplicate a request at another server; this is a facility called The Shadow, which has duplexed databases for reliable systems.

The constraints of small machines and the requirement to be transparent to existing operating systems makes protocol design and implementation a challenging problem. Memory capacities are limited and most operating systems were not designed with data sharing or network interconnection in mind. The client-server protocol designed by Nestar has been implemented at different times on five substantially different hardware bases in four different languages supporting seven different operating systems. It has proven to be an efficient and extendible layer upon which operating system and application processes depend for services.

References: "OSI Reference Model - The ISO model of architecture for open system interconnection", H. Zimmerman, IEEE Transaction on Communications, vol. COM-28, pp. 425-432, April 1980. (A description of the OSI-ISO 7-level model of networked systems.)

"Internet Transport Protocols", Xerox Corp. System Integration Standard, Stanford, Conn.; December 1981; XSIS-028112. (Describes the Network and Transport Levels for XNS, including the Sequenced Packet Protocol.)

"Token-Passing Protocol boosts throughput in local networks", J.A. Murphy, Electronics, June 16, 1981. (An overview of the ARCNET local area network for the physical and datalink levels.)

"Experiences with a Layered Approach to Local Area Network Design", G.M. Ellis, *et al.,* IEEE Selected Areas on Communications, December, 1983. (Nestar's implementation of levels 1-4).

Session protocol is key to X.25 interface

Most packet assemblers/disassemblers perform the X.25 interface function via a monolithic software implementation of CCITT Recommendations X.3, X.28 and X.29. These recommendations define an architecturally independent upper-level protocol that permits the connection of generic ASCII asynchronous terminals to a packet-switched network. This implementation has proved to be adequate because of the relative simplicity of the start/stop protocol and ASCII code used by async terminal devices.

However, a generic PAD function is harder to implement when one must interface terminals or hosts that operate with vendor-specific synchronous protocols, such as BSC-3 (3270), SDLC, Uniscope, IPARS, VIP770 and Burroughs' TD830. These protocols are typified by a complex set of state transitions, terminal format types and control sequences that reflect the master/slave relationship between hosts and terminals.

Protocom approaches this problem by using the seven-layer ISO OSI architecture as a foundation to implement an X.25 interface. Although existing synchronous-protocol specifications also adhere to the OSI model in some ways, most protocol functions refer directly to X.25 operations—and thus implementations tend to be protocol-specific and require the development of a new mapping function (Fig 1). In contrast, the Protocom implementation identifies the generic aspects of the interface by partitioning the interface problem into three logically distinct modules (Fig. 2):

- Application module, which interfaces terminal and host applications with Protocom's generic session module.
- Transport module, which manages the network resources during the transfer of information, and interfaces with the session module.
- Session module, which provides the integration and synchronization of the application and transport modules so that the interface function is performed transparently to network users.

Protocols	Applicable specifications
IBM 3270	D3303 or DSP
IBM SDLC	PRPQ - 5799 ARJ or ANSI HPAD
IBM 2780/3780	DATAPAC BPAD

1. The problem with previous specifications for synchronous protocols is that, unlike the generic approach used in the Protocom X.25 interface, they are protocol-specific. For each specified, protocol, therefore, a new mapping is required (like those shown above).

Rafael Collado, President
Stephen Felner, Technical Support
Protocom Devices
207 Atlantic Street
Stamford, Conn. 06901-3504